Agent-based Dependency-related Build Repair

Christian Macho
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
christian.macho@aau.at

Katharina Stengg
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
katharina.stengg@aau.at

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
martin.pinzger@aau.at

***Abstract*—Building software is a crucial task in today's software projects. Build systems support developers by automating all essential tasks, such as compiling, testing, and assembling software projects. However, they must be maintained regularly to prevent build failures. Neglected maintenance of build configurations remains one of the main causes of build failures.**

In this paper, we explore an agent-based approach to automatically repair broken builds caused by dependency declaration problems. We leverage the capabilities of large language models (LLMs) to closely collaborate with provided tools. Our approach combines tools for retrieving further information about dependencies and build execution results with the LLMs.

In a preliminary evaluation, we show that using the full error log our approach repairs up to 77% of the failing builds with an average of 1.32 repair steps. Using the parsed error log, our approach yields a repair rate up to 83% with 1.27 repair steps on average. In both cases, developers can also use smaller models run on their machine with only small reduction in performance.

***Index Terms*—Automated Build Repair, Dependency Management, Large Language Model, Agent-based Software Engineering**

I. Introduction

Continuous integration (CI) relies heavily on build tools [1], [2] that automate the core tasks of compiling, testing, and packaging software. By handling these repetitive steps automatically, developers can focus on implementing features rather than manual building the software project. Such CI pipelines reveal defects early when the developers are still aware of the code they produced. This allows for faster repairs and, thus, increased developer productivity [3], [4].

While the benefits of using CI are clear, developers also face drawbacks. Hilton *et al.* [5] showed that many benefits come with strade-offs, such as immediate feedback and additional maintenance [6]. Neglecting this maintenance can increase the number of failing builds [7], [8], with dependency-related build breakage among the most common causes. While recent studies highlight the need of build maintenance, build breakage might still occur [7], [9], [10], and manually repairing such breakage is time-consuming and error-prone. To address this, several automated build-repair approaches have been proposed [11], [12], but they typically handle only a selected set of breakage types, such as missing or wrongly declared dependencies.

With the recent advances of LLMs, automated repair received a new direction that might generate new insights to automated build repair and allow to repair more types of build breakage. Similar to other fields [13]–[15] LLM-based, and in particular agent-based, repair shows high potential as a complement to recent approaches for build repair. To that end, we plan to investigate the potential of agent-based dependency-related build repair in this work. We present an approach that uses LLMs to retrieve repair suggestions which are then executed by our build agent. Moreover, we allow the LLM to retrieve information that it needs to solve the problem from a task-specific agent, *e.g.*, retrieving detailed information about a dependency. This motivates our first research question:

(RQ1) To what extent can we repair dependency-related build breakage using our agent-based approach?

We show that our approach can repair up to 77.24% of the studied cases, many of them within a single repair step. We also observe that also smaller LLMs show reasonable performance, repairing up to 68.28% of the cases. While these results are promising, we argue that providing the full error logs might not be needed and cost only extra computing resources and money if commercial LLMs are used. Thus, we investigate the opportunity of only providing parsed error logs from MavenLogAnalyzer (MLA) containing the build result and selected, relevant build details. We address this with our second research question:

(RQ2) To what extent can concise information of error logs help to improve the repair success using our agent-based approach?

Providing the summarized error logs, we observe an increase in the repair rates up to 83.10% for the large model. Contrary, the smaller models did not noteably benefit by the parsed information. In summary, this work makes the following contributions: (1) an empirical investigation of how an agent-based approach can automatically repair dependency-related build breakage, and (2) an analysis of the impact of concise error log information of the input.

The remainder of the paper is organized as follows. Section II situates the paper with respect to related work. Section III explains our approach which is evaluated in Section IV. We discuss potential threats, expected implications, and future work in Section V. Finally, Section VI concludes this paper.

II. Related Work

In this section, we discuss relevant related work to put our work into context. We focus on the closest works on repair broken builds to highlight the difference to our work.

A few works studied dependencies from the point of their ability to break the build. For example, Reyes *et al.* [16] present Breaking-Good, an approach that finds the root cause of breaking dependency updates. It accurately identified 70% of the root causes of the studied cases. Macho *et al.* [17] propose DValidator, a tool that identifies dependency smells in Maven projects. It is capable of detecting five types of dependency smells that might be related with build breakage.

Furthermore, approaches to automatically repair broken builds have been studied frequently. Macho *et al.* [11] studied build-repairing commits to understand and identify how the builds were repaired. With the learned repair strategies, they propose BuildMedic, their tool to repair broken builds. It repairs 45% of the studied broken builds out of which 76% only needed a single dependency modification. Hassan *et al.* [12] proposed HireBuild, which leveraged historical build information to provide build-fixing patches. HireBuild could repair 45% of the investigated cases.

The closest works to the presented approach are the works of Frunkte and Krinke [18] and Reyes *et al.* [19] which both automatically repair build breakage with the use of LLMs. Frunkte and Krinke [18] achieve 23% repair rate on the BUMP dataset [20], and Reyes *et al.* [19] improve the repair rate to 78% by incorporating contextual information.

While these approaches already set a solid baseline for automated build repair, they mainly work with very large models which can be very costly, or impossible to run on a developer machine. Moreover, adding only additional information to a prompt might not substantially improve the repair power of the approaches. In this work, we address these two problems by first, investigating whether full information of the error is needed or if concise error log information suffices, and second, we leverage the agentic capabilities of LLMs to retrieve the information it needs in a focused way.

III. Agent-based Dependency Repair

In this section, we describe how we leverage LLMs to automatically repair errors that occur from wrongly declared dependency configurations. In this paper, we focus on the Apache Maven build tool because it is one of the most used in the Java ecosystem.

A. Approach

Figure 1 shows an overview of our approach. Our approach starts by executing the build in the current version of the project. The execution yields a build outcome, which is either build successful or build failing. In the case of a failing build, the build execution additionally yields the error message of the build. Using this information about the build success and the declared dependencies, our approach creates a prompt that asks an LLM to improve the dependency configuration. In case the LLM needs more information, the approach provides specific questions that it passes on to specialized tools which will answer the questions and feed back the answers to the LLM. As the last step, our approach executes the actions that the LLM provides, for example adding a dependency

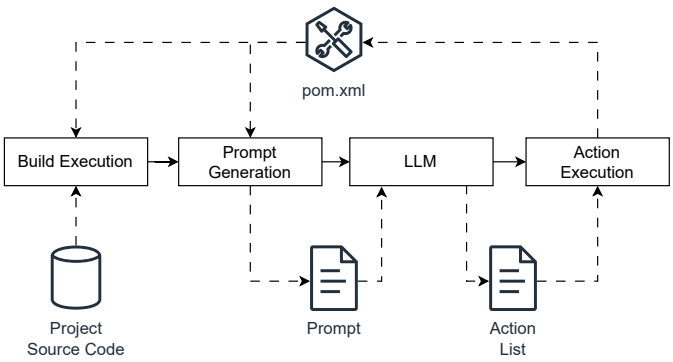

Fig. 1: Approach of one repair step. This approach is repeated until the build is successful or the maximum number of steps is exceeded.

in the pom.xml file. This loop is repeated until the build outcome becomes successful or the maximum number of tries is exceeded. In the following, we explain each step in detail.

Build Execution. As the first step, our approach retrieves the build outcome from the current version of the project by executing the Maven build system using the following call: `mvn clean package -DskipTests=true -U`. The Maven goal `clean` deletes all artifacts of previous builds to ensure that we build upon a clean sheet. The goal `package` invokes the compilation, dependency resolution, and assembly of the project. We decided to exclude test execution, using the `-DskipTests=true` toggle, because first, we focus on repairing dependency-related failures, and second, tests might not only fail due to technical reasons, but also for various other reasons, such as flaky tests or environmental impact [21], [22] which are out of scope for this study. Finally, the `-U` toggle forces the build system to check for missing local dependencies every time the build is executed. Our approach then uses the MavenLogAnalyzer (MLA) [11], [23] to extract the actual build outcome and failure details from the full execution logs.

Prompt Generation. As described before, we leverage the capabilities of LLMs to output repair suggestions for repairing a broken build. LLMs are usually controlled by providing prompts which should be carefully created [24] as LLMs are sensitive to variations in the prompts and even the role that the LLM should take [25].

Generally, the communication with an LLM is started with a so-called system prompt. Our system prompt as seen in Listing 1 sets the context for all further communication. Recent literature suggests that such a context should be provided to improve the results of the tasks [25]. In our case, the system prompt tells the LLM that it should act as a senior developer who is also expert in the Apache Maven ecosystem. Moreover, it provides several rules that the LLM must consider when answering to any further prompts. In our case, these rules refer to well-known rules for dependency management and generic advices when selecting dependencies for projects.

Our approach then creates a user prompt for each repair step that contains detailed information about the current build failure as shown in Listing 2. The user prompt first contains

Listing 1: System Prompt

```
You are a senior developer and an expert using Apache Maven and its ecosystem, in particular with dependency management.
    ↪ You are helping me fix build issues. I will provide you with build errors and the list of currently declared
    ↪ dependencies. Your task is to output a list of dependencies that make the build succeed. Follow these rules:
- You cannot ADD or UPDATE a dependency to com.lob:lob-java, because this is that project.
- Suggest only dependencies that are strictly required to resolve the build error.
- Leave dependencies untouched and unmodified that you do not need to modify and also return them in the final list.
- Match dependency versions to be compatible with the current environment (e.g., Java version, framework version).
- If multiple versions exist, only choose stable (non-beta, non-alpha) version unless the code requires a specific one.
- If multiple versions exist, choose the most recent ones if possible.
- If multiple versions exist, avoid versions older than 3 years.
- Do not suggest duplicates - check if a dependency (or an equivalent via transitive dependencies) is already included.
- Provide a list of the dependencies that should be declared in the format groupId:artifactId:version:scope including the
    ↪ ones you leave untouched and nothing else in your response. If no scope is specified in the pom, use "compile".
- Only output this list, nothing more.- If you are uncertain, suggest the closest likely option rather than leaving it
    ↪ blank.
```

Listing 2: User Prompt Template

```
Your job is to examine the dependencies of a Maven pom.xml and suggest the minimal set of changes that will fix the
    ↪ reported problems.
Strictly follow all rules from above and below.
You must output **only** one of the following command styles per line:

* `ADD groupId:artifactId:version`
* `REMOVE groupId:artifactId:version`
* `UPDATE oldGroupId:oldArtifactId:oldVersion` to groupId:artifactId:version (for replacing an existing element)

If you are unsure that a dependency exists, for example, when the error message indicates so, you can ask me the following
    ↪  questions:
* `WHICH_ARTIFACTS_EXIST: groupId` (--> if you need to know which artifacts exist of a certain groupID)
* `WHICH_VERSIONS_EXIST: groupdId:artifactId` (--> if you need to know which versions of an artifact exist)

Example valid outputs:
ADD com.example:myapp:1.0.0
REMOVE org.springframework:spring-core:5.3.0
UPDATE com.example:oldlib:1.0.0 to com.example:newlib:2.0.0
WHICH_ARTIFACTS_EXIST: com.example
WHICH_VERSIONS_EXIST: com.example:myapp

No explanatory text, no extra formatting, no markdown, only one of the above mentioned.
If an element is already correct, do not output anything for it.
Here are the current dependencies of the pom.xml:
[dependencies will be listed here]

Here is the current output:
[build output details are provided here]
```

an instruction that the LLM should aim to repair the presented problem. Moreover, the prompt provides specific instructions what the LLM is allowed to return. In particular, it must either return a list of actions that modify the dependency list (*ADD*, *REMOVE*, or *UPDATE*), which our approach executes in a later step of our approach, or the LLM can ask for more information concerning the existence of artifacts. First, it may ask which artifacts exists for a given groupId (*WHICH_ARTIFACTS_EXIST*). A task-specific agent will then look up the list of artifacts for the given groupId in the Maven Central repository[1] and return the retrieved list to the LLM. Second, it can ask for a list of available versions of a given groupId and artifactId (*WHICH_VERSIONS_EXIST*). Similarly, our approach leverages another task-specific agent that retrieves the information from the Maven Central repository and provides the list of available versions to the LLM. Both questions help the LLM to avoid creating actions for dependencies that might not exist, for example, adding a dependency that looks similar to the correct one, but does not exist.[2] The prompt also provides the list of dependencies that are currently declared in the `pom.xml` so that the LLM can incorporate the declared dependencies in the decision of which action might be the best next step.

Furthermore, the prompt contains a detailed error description. In our experiments, we evaluate two strategies to provide such a description. First, the prompt provides the full error message that Maven outputs. In particular, this concerns the error details below the build result in the Maven output. Second, our approach parses the full log using MAVENLOGANALYZER [11], [23] and only provides the build result alongside error details such as a list of missing packages or missing classes in the prompt. That way, the prompt is more concise and might pinpoint the LLM to repair the build failure more often and faster. Moreover, shorter prompts are typically cheaper when using a LLM.

Action Execution. This step retrieves the action list provided by the LLM and executes each step. For *ADD* actions, it adds the specific dependency in the listed version to the `pom.xml`, for the *REMOVE* actions, it removes the respective

[1]https://repo1.maven.org/maven2/{groupId}

[2]For example, in one experiment case, the LLM planned to add a dependency called "sauce-rest" but in fact, it is called "saucerest". This was corrected by the LLM after providing the artifacts of groupId "com.saucelabs"

declaration, and for *UPDATE* actions, it identifies the affected dependency declaration using the old Maven coordinates, and replaces them with the new coordinates provided in the action list. The actions are executed in the order that the LLM outputs them.

Finally, our approach re-executes the build with the updated configuration to verify whether the dependency changes succeeded, *i.e.*, led to a repaired build. If not, the process repeats until the maximum number of steps is reached.

IV. Preliminary Evaluation

In this section, we present the results of our preliminary evaluation. We first describe the setup that we used for the evaluation and then present the preliminary results of this study.

1) Setup: In the following, we give insights on how we set up the evaluation environment and which decisions we made for the evaluation.

Project Selection. For this ERA work, we decided to use Apache Maven open-source Java projects that leverage the Maven dependency ecosystem, *i.e.*, they declare at least three dependencies additional to their test dependencies. Moreover, we selected projects of different vendors and different size to mitigate bias.

LLM Selection. For this preliminary evaluation, we restrict ourselves to three open-source of LLMs, *i.e.*, **gpt-oss:20b, gpt-oss:120B, and qwen3-coder:30b**.We choose the three as they are known to work well in coding tasks and are also publicly available. Moreover, running these variants of LLMs is feasible on a small-sized server or, in the case of gpt-oss:20b and qwen3-coder:30b, even on a developer machine. We also experimented with other LLMs, such as gemma3:27b, codellama:70b, llama3.3:70b, and phi4:14b, but they all ignored the instruction to only provide the five provided actions as a response and output full-text instructions instead. We are aware that recent research revealed that the prompts should be tailored to the specific LLM [26], however, we decided to leave this to future work, and focus on the described models for this ERA paper to provide a preliminary proof of concept for agent-based build repair. Moreover, we include two smaller versions and one larger version of LLMs to investigate the potential impact of the model size on our approach and to evaluate the possibility to run such models locally on the developers computer.

Evaluation Approach. For this evaluation, we set the maximum number of repair steps to ten, which is a compromise between faster feedback from the tool and allowing our approach to repair a scenario not only within a single step, but within several steps.

For each of the selected projects, we identify all non-test dependencies that the project declares in the studied revision. Next, for each dependency, we create a failure scenario in which this dependency is removed from the dependency list. This causes a missing dependency failure when building the project. Additionally, we exclude the removed dependency from the remaining declared dependencies by adding an exclusion in the `pom.xml`. This is necessary because the build would otherwise succeed if the removed dependency is included as a transitive dependency of any other declared dependency. Next, we applied our approach to each failure scenario and evaluate the repair rates. Please note that we configure each LLM to minimize the hallucination effect. We set the temperature parameter to 0.0 which limits the creativity of the LLM to a minimum. Allthough this helps to reduce the effect of creativity in LLMs, it is not guaranteed that the LLMs behave deterministic. Thus, we execute each scenario ten times, and study the repair outcomes of all runs.

TABLE I: Repair Results by Project using the full error log.

Project	gpt-oss:120b	gpt-oss:20b	qwen3-coder:30b
lob-java	**59/70 (84.29%)**	43/70 (61.43%)	48/70 (68.57%)
ci-sauce	**59/60 (98.33%)**	53/60 (88.33%)	47/60 (78.33%)
pdfbox	**20/30 (66.67%)**	12/30 (40.00%)	14/30 (46.67%)
cxf	**25/30 (83.33%)**	13/30 (43.33%)	18/30 (60.00%)
guava-guava	21/50 (42.00%)	15/50 (30.00%)	**30/50 (60.00%)**
zulip-client	10/20 (50.00%)	10/20 (50.00%)	**18/20 (90.00%)**
jackson-databind	**10/10 (100.00%)**	**10/10 (100.00%)**	6/10 (60.00%)
rabbitmq-jms-client	**20/20 (100.00%)**	18/20 (90.00%)	17/20 (85.00%)
Total	**224/290 (77.24%)**	174/290 (60.00%)	198/290 (68.28%)

2) Results: Table I shows the results per project when running our approach with the full error log. We obtain the best score with the gpt-oss:120b model which repaired 77.24% of the studied cases. While the smaller models perform worse than gpt-oss:120b, they still yield a solid repair rate with 60% and 68.28% for gpt-oss:20b and qwen3-coder:30b, respectively. Considering that these two models are four times and six times smaller than gpt-oss:120b, we understand this as a solid performance. Considering the projects in detail, we see that all models can repair at least some of the studied cases. Allthough gpt-oss:120b is the best performing model overall, qwen3-coder:30b provides the highest minimum repair rate per project. However, it never fixes all cases of a studied project.

We also study the number of repair steps that are needed to repair a case. We observe that a successful repair is achieved after 1.32, 1.49, and 1,26 steps for the three models, respectively. These results indicate that all models can repair most of the problems within a few steps or even with a single repair step. However, the results also show that our approach is capable of repairing cases that cannot be repaired within one step.

To that end, we can answer RQ1: **To what extent can we repair dependency-related build breakage using our agent-based approach?**

> Using the full error log, our approach can repair up to 77.24% of the failing cases on average. We also observe that smaller models yield a solid performance up to 68.28% on average. Moreover, many cases can be repaired with only one or a few repair steps.

Table II shows the results for the experiment using the parsed build log. We see that again the gpt-oss:120b model outperforms the other studied models reaching an overall repair rate of 83.10%. This indicates that the parsed in-

TABLE II: Repair Results by Project using the parsed error log.

Project	gpt-oss:120b	gpt-oss:20b	qwen3-coder:30b
lob-java	**60/70 (85.71%)**	47/70 (67.14%)	42/70 (60.00%)
ci-sauce	**60/60 (100.00%)**	52/60 (86.67%)	40/60 (66.67%)
pdfbox	**20/30 (66.67%)**	10/30 (33.33%)	0/30 (0.00%)
cxf	**30/30 (100.00%)**	13/30 (43.33%)	7/30 (23.33%)
guava-guava	21/50 (42.00%)	22/50 (44.00%)	**25/50 (50.00%)**
zulip-client	**20/20 (100.00%)**	13/20 (65.00%)	**20/20 (100.00%)**
jackson-databind	**10/10 (100.00%)**	**10/10 (100.00%)**	7/10 (70.00%)
rabbitmq-jms-client	**20/20 (100.00%)**	14/20 (70.00%)	1/20 (5.00%)
Total	**241/290 (83.10%)**	181/290 (62.41%)	142/290 (48.97%)

formation helps the model to achieve better results when repairing dependency-related build breakage. However, the smaller models did not improve their performance equally. While the gpt-oss:20b model still shows an overall small increase to 62.41% successfully repaired cases, the qwen3-coder:30b model performed notably worse with the parsed build log as input, obtaining a decreased repair rate from 68.28% to 48.92%.

Again, we also study the number of steps that are needed for a successful repair. Using the parsed build log, we see that the repair was successful after 1.27, 1.53, and 1.09 steps, for the three models, respectively. Similar to RQ1, most of the successful repairs are achieved within a few steps or a single step. Furthermore, we also observe cases that were repaired with more than a few steps.

With these results, we answer RQ2: **To what extent can concise information of error logs help to improve the repair success using our agent-based approach?**

> Using the parsed error log, our approach can repair up to 83.10% of the failing cases on average. The smaller models did not benefit notably from the parsed error log. The qwen3-coder:30b model even notably decreased in performance. Moreover, we again see that many cases can be repaired with only one or a few repair steps.

V. Discussion

In this section, we first explain how we mitigated the threats to validity. Next, we highlight selected implications that our work might pose on research and industry. Finally, we outline future research directions posed by this work.

A. Threats to Validity

A threat to **internal validity** concerns the exclusion of the tests. Our approach might perform differently if we also provided data from a test run. However, in this ERA work, we solely focus on dependency-related breakage, which mitigates the threat. Another threat relates to the selection of LLMs in this study. We only incorporated three LLMs from two vendors. However, as mentioned earlier, we experimented with other models from other vendors as well, but they did not follow the requested output format properly. Lastly, we only use one prompt template in this work and abstain from creating customized prompts for each model. However, the used models in this paper worked well with the provided prompt.

Concerning **external validity**, we identified the project selection as a threat. Currently, we only selected eight suitable projects and did not conduct a large-scale evaluation with more projects. We mitigated this threat by carefully selecting the projects and ensuring that they are from different vendors, size, and purpose. We are aware that the results of this study might not generalize for that reason.

B. Implications

Our work has implications both on research and developers. Concerning **research**, we showcase that LLMs are capable of repairing dependency-related build breakage without task-specific fine-tuning. Our results indicate that, when supplied with the appropriate tools, LLMs are an excellent complement to state-of-the-art build repair techniques, such as the approaches by Macho *et al.* [11] or Hassan *et al.* [12].

The implications for **developers** mainly concern their productivity. Using our approach to repair dependency-related build breakage, allows developers to focus on their actual work, while repair suggestions for a failing build are automatically provided. Hence, we hypothesize that developers might increase their overall productivity for that reason.

C. Future Work

We foresee several possible future research directions. First, we suggest to enrich the input for the LLMs with additional information, for example supplying more detailed information about the environment or the project itself. Second, we envision that incorporating additional tooling (*e.g.*, build tools such as DVALIDATOR [17]) will help to repair more cases successfully. Moreover, we also envision that it will decrease the number of necessary repair steps by reaching a successful build faster with more detailed and diverse information provided. Lastly, we expect a high potential for build repair when our approach fully adopts an agentic approach. Leaving decisions to LLMs which repair steps to take and which tools to query, we expect that even build that were unsuccessfully repaired might become automatically repairable.

VI. Conclusion

In this paper, we presented and preliminary evaluated our approach that leverages the agentic features of LLMs to repair broken builds. We showed that using two variants of input, our proposed approach can repair up to 77.24% of the studied cases when using the full build log and up to 83.10% when using the parsed build log as input. We anticipate that our approach can lay the foundation for further approaches that leverage LLMs for automated build repair.

Acknowledgment

This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/P36698. For open access purposes, the author has applied a CC BY public copyright license to any author accepted manuscript version arising from this submission. The research was supported by the Austrian ministries BMIMI, BMWET and the State of Upper Austria in the frame of the SCCH COMET competence center INTEGRATE (FFG 892418).

References

[1] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *Proceedings of the International Conference on Software Maintenance and Evolution*. New York, NY, USA: IEEE, 2014, pp. 401–405.

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the International Conference on Automated Software Engineering*, New York, NY, USA, 2016, pp. 426–437.

[3] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2015, pp. 805–816.

[4] B. Adams and S. McIntosh, "Modern release engineering in a nutshell: Why researchers should care," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, New York, NY, USA, 2016, pp. 78–90.

[5] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 197–207.

[6] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter, "The evolution of the linux build system," *Electronic Communication of the European Association of Software Science and Technology*, vol. 8, 2007.

[7] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *International Conference on Software Engineering*. ACM, 2014, pp. 724–734.

[8] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 41–50.

[9] M. Sulír and J. Poruban, "A quantitative study of java software buildability," in *Proceedings of the International Workshop on Evaluation and Usability of Programming Languages and Tools*, 2016, pp. 17–25.

[10] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, 2017.

[11] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2018, p. 106–117.

[12] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1078–1089.

[13] A. Almeida, L. Xavier, and M. T. Valente, "Automatic library migration using large language models: First results," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 427–433.

[14] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1646–1656. [Online]. Available: https://doi.org/10.1145/3611643.3613892

[15] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1482–1494.

[16] F. Reyes, B. Baudry, and M. Monperrus, "Breaking-good: Explaining breaking dependency updates with build analysis," in *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2024, pp. 36–46.

[17] C. Macho, F. Oraze, and M. Pinzger, "Dvalidator: An approach for validating dependencies in build configurations," *Journal of Systems and Software*, vol. 209, p. 111916, 2024.

[18] L. Fruntke and J. Krinke, "Automatically fixing dependency breaking changes," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2146–2168, 2025.

[19] F. Reyes, M. Mahmoud, F. Bono, S. Nadi, B. Baudry, and M. Monperrus, "Byam: Fixing breaking dependency updates with large language models," *arXiv preprint arXiv:2505.07522*, 2025.

[20] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, and M. Monperrus, "Bump: A benchmark of reproducible breaking dependency updates," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 159–170.

[21] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840.

[22] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 101–111.

[23] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: An empirical study of travis ci," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2018, p. 87–97.

[24] Q. Ma, W. Peng, C. Yang, H. Shen, K. Koedinger, and T. Wu, "What should we engineer in prompts? training humans in requirement-driven llm use," *ACM Transactions on Computer-Human Interaction*, vol. 32, no. 4, pp. 1–27, 2025.

[25] L. Salewski, S. Alaniz, I. Rio-Torto, E. Schulz, and Z. Akata, "In-context impersonation reveals large language models' strengths and biases," *Advances in neural information processing systems*, vol. 36, pp. 72 044–72 057, 2023.

[26] G. Wang, Z. Sun, S. Ye, Z. Gong, Y. Chen, Y. Zhao, Q. Liang, and D. Hao, "Do advanced language models eliminate the need for prompt engineering in software engineering?" *ACM Transactions on Software Engineering and Methodology*, 2024.